Algorithmique ~~et Programmation~~
TD n° 1 : Introduction

Ecole normale supérieure
Département d'informatique
td-algo@di.ens.fr

2011-2012

1 Mode d'emploi

Les exercices 1, 8 et 12 illustrent la technique du "diviser pour régner" avec un niveau de sophistication croissant. Les exercices 4, 3, 9 et 10 sont consacrés à la méthode gloutonne, et enfin les exercices 2, 5, 6 et 7 sont consacrés à la méthode de la "programmation dynamique". L'exercice 11 est inclassable, et ne nécessite aucun prérequis. L'exercice 13 est consacré à la preuve de terminaison d'un algorithme (en apparence) simple.

Nous ne ferons pas de programmation ensemble, mais l'exercice 8 fournit des fonctions intéressantes dont la programmation n'est pas entièrement triviale.

2 Petits Rappels

diviser pour régner Cette stratégie résout un problème en le découpant en sous-problèmes du même type, en résolvant récursivement les sous-problèmes, et en combinant les sous-résultats de façon appropriée. Cela aboutit typiquement à un fonctionnement "en arbre".

programmation dynamique C'est une généralisation du "diviser pour régner". Cette méthode a pris son nom dans les années 1950, quand le mot "programmation" signifiait tout autre chose qu'aujourd'hui. On identifie également une collection de sous-problèmes, et on les résoud un par un, en commençant par les plus petits, en utilisant les solutions des petits sous-problèmes pour résoudre les problèmes plus gros, jusqu'à ce qu'ils soient tous résolus. La différence avec l'approche "divide and conquer" de base est que le processus n'est pas décrit par un arbre, mais par un graphe orienté acyclique (la solution d'un sous-problème peut re-servir plus d'une fois dans la suite).

algorithmes gloutons Ces méthodes construisent généralement une solution morceau par morceau, et en choisissant le prochain morceau, elles choisissent systématiquement celui qui procure l'avantage le plus évident et le plus immédiat.

Mots x et z sont, respectivement, un préfixe et un suffixe d'une chaîne de caractères ω si on peut écrire $\omega = x.z$. De même, y est un facteur de ω si on peut écrire $\omega = x.y.z$, où x et z sont éventuellement vides.

Approximation Etant donné un problème d'optimisation, une λ-approximation est un algorithme polynomial qui produit des solutions au pire λ fois plus grandes (resp. plus petites) que la meilleure.

Graphes Un graphe (non-orienté) $G = (V, E)$ est la donnée d'un ensemble de sommets V (pour "vertices", pluriel irrégulier de "vertex"), et d'un ensemble d'arêtes $E \subseteq V^2$ reliant certains sommets. En gros, $x \leftrightarrow y$ dans G ssi $(x, y) \in E$. Le graphe complémentaire de G, noté $\overline{G}$ est obtenu en mettant des arêtes là où il n'y en avait pas et réciproquement. L'union de deux graphes dont les ensembles de sommets sont disjoints est bien définie (on met les deux graphes côtes-à-côte). Un graphe est dit connexe si ce n'est pas l'union de deux graphes (c'est-à-dire s'il est "d'un seul tenant").

Logique Une variable booléenne prend comme valeur "vrai" ($\top$) ou "faux" ($\bot$). L'ensemble des formules booléennes contient les variables, $\bot$, $\top$, et il est fermé pour la négation ($\overline{\psi}$), le ET ($\phi \wedge \psi$) ainsi que le OU ($\phi \vee \psi$). Une formule est satisfiable s'il existe une assignation des variables qui la rend vraie. Sinon, elle est insatisfiable. Un littéral est soit une variable, soit la négation d'une variable. Une formule booléenne est en forme normale conjonctive Si elle peut s'écrire

$$\psi = \bigwedge_i \bigvee_j \ell_{ij}$$

où ℓ_{ij} est un littéral.

3 Exercices pour le TD

Exercice 1 (⋆) : Fragile. On dispose d'un stock de boules en verre. Le problème est de déterminer à partir de quel étage d'un immeuble les boules en verre se cassent si on les jette par la fenêtre. Vous êtes dans un immeuble à n étages (numérotés de 1 à n) et vous disposez de k boules. Il n'y a qu'une seule opération possible pour tester si la hauteur d'un étage est fatale : jeter une boule par la fenêtre. Si elle ne se casse pas, vous pouvez la réutiliser ensuite, sinon vous ne pouvez plus.

Vous devez proposer un algorithme pour trouver la hauteur à partir de laquelle un saut est fatal (renvoyer $n+1$ si on survit encore au n-ième étage) en faisant le minimum de sauts.

1. Si $k \geq \lceil \log_2(n) \rceil$, proposer un algorithme en $\mathcal{O}(\log_2(n))$ sauts.
2. Si $k < \lceil \log_2(n) \rceil$, proposer un algorithme en $\mathcal{O}\left(k + \frac{n}{2^{k-1}}\right)$ sauts.

(⋆) 3. Si $k = 2$, proposer un algorithme en $2\sqrt{n}$ sauts.

(⋆⋆) 4. Dans ce dernier cas, proposer aussi un algorithme en $\sqrt{2n}$ sauts.

Exercice 2 (⋆⋆) : Fragile, le retour. Décrivez un algorithme qui étant donné n et k renvoie le nombre minimal de lancers à effectuer pour être sûr de trouver la solution. Déterminer sa complexité.

Exercice 3 (⋆⋆) : Set-Cover. On considère un ensemble fini B à n éléments, et une famille finie $S_1, \ldots, S_m$ de sous-ensembles de B. Le problème Set-Cover est de déterminer le plus petit (au sens de la cardinalité) ensemble $I \subseteq 1, \ldots, m$ tel que :

$$B \subseteq \bigcup_{i \in I} S_i$$

1. Donnez un algorithme glouton. Démontrez qu'il n'est pas optimal.

(⋆⋆) 2. Démontrez ensuite que si $|B| = n$, alors il produit une solution qui est au pire $\ln n$ fois plus grande que l'optimale. (Indice : pour ça, on peut prouver quelque chose sur le nombre de points couverts par chaque choix glouton de l'algorithme).

3. Démontrer que le pire cas est atteint (asymptotiquement).

4 Exercices à faire chez vous

Exercice 4 (⋆) : Problème du voyageur de commerce. On considère une carte géographique sur laquelle figurent n villes, et un voyageur de commerce qui doit visiter toutes ces villes une fois (et une seule). Le voyageur se déplace en avion, en ligne droite. Le problème consiste, en partant d'une des villes, à trouver un ordre dans lequel visiter toutes les autres villes qui minimise le nombre de kilomètres parcourus.

Proposez 3 algorithmes gloutons qui donnent des solutions *approximatives* au problème (on ne demande pas de prouver la qualité des solutions). Il doit être possible (ce n'est pas obligatoire) d'exhiber des exemples qui montrent que vos algorithmes ne sont pas des λ-approximations si λ est une constante indépendante de n.

Exercice 5 (⋆⋆) : Algorithme de Kadane. On se donne un tableau A de n éléments entiers $A[1], \ldots, A[n]$. Le problème est de déterminer deux indices $1 \leq i, j \leq n$ tels que la somme $A[i] + \cdots + A[j]$ est maximale. Notez que $j < i$ est une solution admissible (et alors la somme vaut zéro). Donnez un algorithme (programmation dynamique) qui n'examine chaque case de A qu'une seule fois (et qui termine donc en temps linéaire).

Exercice 6 (⋆⋆) : Plus long facteur commun. Le problème est de déterminer quelle est la longueur du plus long facteur commun à deux chaînes de caractères données. Donnez un algorithme (programmation dynamique) et évaluez sa complexité.

Exercice 7 (⋆⋆) : Massacre à la tronçonneuse. Admettons qu'on ait affaire à un langage de programmation où la seule opération possible sur les chaînes de caractère soit une fonction Split, qui prend en argument une chaîne de caractère ω de taille n et un entier k, et qui renvoie deux chaînes : un préfixe de taille k de ω ainsi qu'un suffixe de taille $n-k$. Cette procédure s'exécute inévitablement en temps n. On ne peut pas lire les "cases" individuelles des chaînes (!).

Maintenant, on veut fabriquer une fonction MultiSplit qui prend en argument une chaîne de caractères ω (de taille n) et une liste $[a_1; \ldots, a_m]$ d'entiers, et qui renvoie les $m+1$ chaînes :

$$\omega[1..a_1]; \omega[a_1..a_2]; \ldots; \omega[a_{m-1}..a_m]; \omega[a_m..n].$$

1. Donnez une version récursive naïve de MultiSplit, ainsi que son temps d'exécution.
2. Donnez ensuite un algorithme qui détermine dans quel ordre effectuer les coupes pour minimiser le temps d'exécution (programmation dynamique). Est-il plus rapide que la méthode naïve ?

Exercice 8 (⋆⋆⋆) : 3SAT. On s'intéresse au problème 3SAT, qui consiste à déterminer si une une formule logique en forme normale conjonctive avec au plus trois littéraux par clause est satisfiable. On considère une formule logique ψ (au format 3SAT). On va noter $\psi|x$ la formule ψ dans laquelle on remplace le littéral x par $\top$.

1. Démontrer que si $\psi' \wedge x$ est insatisfiable, alors toute assignation des variables satisfaisant ψ contiendrait $\overline{x}$.
2. Démontrez si x est un littéral, alors ψ et $(\psi \wedge x) \vee (\psi \wedge \overline{x})$ sont équivalentes (c.a.d. ont les mêmes modèles). Déduisez-en un algorithme récursif pour 3SAT de complexité $\mathcal{O}\left(Poly(n) \cdot 2^n\right)$.

On va maintenant chercher à obtenir un algorithme asymptotiquement plus rapide. Si ψ n'est pas vide, alors elle s'écrit $\psi = (x \vee y \vee z) \wedge \psi'$, où x, y et z sont des littéraux.

(⋆⋆) 3. Démontrez que ψ est équivalente à

$$(x \wedge \psi') \vee (y \wedge \psi') \vee (z \wedge \psi')$$

Déduisez-en un algorithme récursif de complexité $\mathcal{O}\left(1.8392^n\right)$ (indice : il y a 3 appels récursifs).

(⋆⋆) 4. Un littéral x est dit pur si $\overline{x}$ n'apparaît pas dans ψ. Démontrez que si ψ est satisfiable, alors il existe une assignation des variables dans laquelle tous les littéraux purs sont vrais. Cela permet de se ramener au cas où aucun littéral n'est pur. Cela signifie que si ψ est non-triviale, on peut l'écrire :

$$\psi = (x \vee y \vee z) \wedge (\overline{x} \vee u \vee v) \wedge \psi'$$

où u et v sont des littéraux arbitraires (qui peuvent très bien être $y, \overline{y}, z$ et $\overline{z}$). Déduisez-en un algorithme récursif de complexité $\mathcal{O}\left(1.7692^n\right)$ (indice : il y a 4 appels récursifs).

(⋆⋆⋆) 5. Justifiez que dans chaque appel récursif, ou bien on a fait apparaître un littéral pur (qu'on peut donc éliminer), ou bien on fait apparaître une clause à deux littéraux. Déduisez-en un algorithme de complexité $\mathcal{O}\left(1.6180^n\right)$ (c'est le nombre d'or !).

(⋆⋆⋆⋆⋆) 6. Retrouver l'algorithme déterministe dont le pire-cas est le moins mauvais connu en 2011 : $\mathcal{O}\left(1.33334^n\right)$.

Exercice 9 (⋆⋆⋆) : Planning de la salle. La salle INFO 4 est très demandée. Tous les départements veulent y faire cours, et tous envoient à l'administration des requêtes de la forme $[s_i; t_i[$, pour utiliser la salle entre l'heure s_i et l'heure t_i (on peut supposer que les heures sont exprimées en secondes depuis le 1er Janvier 1970). L'administration, elle, cherche à rejeter le moins requêtes possible, sachant que deux départements ne peuvent pas faire cours en même temps. Donnez un algorithme glouton pour l'administration, prouvez son optimalité et étudiez sa complexité. Attention, il ne s'agit *pas* de maximiser le nombre d'heure d'utilisation de la salle (mais vous pouvez *aussi* essayer de résoudre ce problème).

Exercice 10 (⋆⋆⋆) : Déménagement. Je vais bientôt déménager et je dois ranger mes affaires dans des cartons. Plus précisément, j'ai à ma disposition un stock illimité de cartons d'un mètre-cube, et je possède n objets dont les volumes sont des rationnels $a_1, \ldots, a_n$ compris entre 0 et 1. Le problème est de trouver un rangement qui occupe le moins de cartons possible (c'est le problème du Bin-Packing).

1. L'algorithme Next Fit est le suivant :
   – prendre les objets dans un ordre quelconque
   – placer l'objet courant dans le dernier carton utilisé s'il tient, sinon fermer le carton en cours et en créer un nouveau

   Montrer que Next Fit est une 2-approximation, et montrer que la borne est atteinte.
2. L'algorithme First Fit Decreasing est le suivant :
   – Trier les a_i par ordre décroissant de volume
   – Placer l'objet courant dans le premier carton utilisé où il tient, sinon en créer un nouveau

   Montrer que First Fit Decreasing est (au moins) une 3/2-approximation.

Exercice 11 (⋆⋆⋆) : Primaires. Comme c'est à la mode, le Parti Pour l'Algorithmique (PPA) organise ses primaires. Les statuts du parti précisent que le candidat qui va être désigné doit avoir obtenu la majorité absolue des voix des membres du parti.

Il y a k candidats, et la totalité des n membres tiennent une réunion dans une église pour élire directement leur candidat, à main levée, par un vote majoritaire à un seul tour. Deux issues sont possibles : ou bien un candidat récolte plus de 50% des voix et il est désigné, ou bien le parti, incapable de se mettre d'accord, se dissout dans la confusion. Seulement voilà, tout le monde à peur de se griller par le vote à main levée ("mais si je vote pour A et que B est élu, B va m'en vouloir à mort, il va me mettre au placard, me coller le TD d'algo, etc."). Il n'a malheureusement pas été prévu de bulletins de vote pour organiser un vote secret.

Pour contourner le blocage qui s'annonce, le président de séance, qui a de la ressource, envisage de mettre les présents à la queue-leu-leu, puis de s'installer dans le confessionnal, et de les faire entrer tour à tour pour demander à chacun son candidat de manière anonyme et confidentielle.

Malheureusement, le président de séance, qui est aussi le doyen du parti, est si vieux que sa mémoire flanche un peu. Il ne dispose plus, à son âge canonique, que de $\lceil \log_2 n \rceil + \lceil \log_2 k \rceil$ bits de stockage ! (Ses bits sont cependant très fiables). Il réalise donc qu'il lui est impossible d'exécuter son plan, car il ne peut pas stocker les $k \cdot \lceil \log_2 n \rceil$ bits qui seraient nécessaire au décompte des scores individuels des candidats.

Il ne peut même pas mesure les score de chaque candidats séparément et garder le max, car cela nécessiterait au moins $2\lceil \log_2 n \rceil + \lceil \log_2 k \rceil$ bits de stockage.

Seulement, comme ce n'est pas le président de séance pour rien, il conçoit un algorithme qui va lui permettre de savoir, malgré ses ressources limitées, si un candidat a la majorité absolue, et si oui, lequel. Comment peut-il faire ?

Exercice 12 (⋆⋆⋆) : Max-Clique. Dans un graphe, une clique est un ensemble de sommets tous mutuellement adjacents : $\forall x, y \in C, (x \neq y) \Rightarrow x \leftrightarrow y$ dans G. On ne connaît pas d'algorithme polynomial pour le problème Max-Clique, qui consiste à déterminer la taille de la plus grande clique présente dans un graphe arbitraire donné en argument. On peut démontrer (vous allez le faire un jour) que le problème est NP-complet, c'est-à-dire qu'il est de complexité comparable à celle de 3SAT.

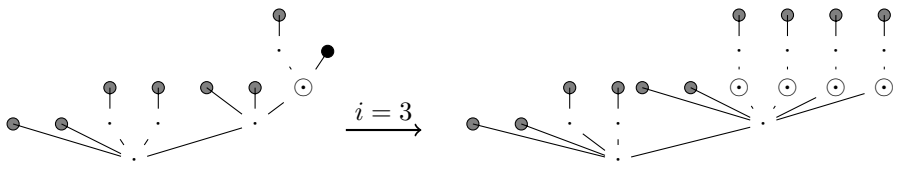
1. Donnez un algorithme pour Max-Clique. Déterminez sa complexité.

On va voir maintenant un cas particulier où un algorithme polynomial existe. Si l'union de deux graphes G_1 et G_2 est bien définie, leur produit, $G_1 \bowtie G_2 = \overline{\overline{G_1} \cup \overline{G_2}}$ l'est également.

2. Montrez que $G_1 \bowtie G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{x \leftrightarrow y \mid x \in V_1, y \in V_2\})$
3. L'ensemble des cographes est le plus petit ensemble de graphes contenant les sommets isolés et fermé par union disjointe et complémentaire. On va construire un algorithme polynomial qui résoud Max-Clique sur les cographes
   a) Montrez qu'à partir d'un cographe, on peut construire un arbre de dérivation qui décrit entièrement le cographe en temps polynomial.
   (⋆⋆⋆) b) Montrez qu'on peut remplacer "complémentaire" par "produit" dans la définition des cographes.
   (⋆⋆⋆) c) Donnez un algorithme qui résout Max-Clique en temps polynomial étant donné une représentation arborescente du cographe pour la nouvelle définition.

(⋆⋆⋆⋆⋆) 4. Donnez un algorithme *linéaire en la taille du graphe.*

Exercice 13 (⋆⋆⋆⋆) : Hercule vs. Hydre. Hercule se bat contre l'Hydre, qui est un monstre à plusieurs têtes. A chaque tour de jeu, Hercule peut couper une des têtes, mais alors l'Hydre a des têtes qui repoussent. Plus précisément, l'Hydre est commodément représentée par un arbre dont les feuilles sont les têtes. En coupant une tête, Hercule détruit une feuille. Il réussit à anéantir le monstre s'il parvient à le réduire à l'arbre vide. Les têtes repoussent de la façon suivante : si la feuille coupée a un grand-père dans l'arbre, alors une fois que la feuille est coupée, son père est cloné i fois, où i est le numéro du tour courant. Par exemple, si on coupe la feuille noire au troisième tour :



Démontrez rigoureusement qu'il est possible de tuer l'Hydre. Essayez (bonne chance) de donner une borne sur le temps d'exécution de l'Hydre (no pun intended).

5 Solutions

Solution de l'exercice 1

1. Il suffit de faire une recherche dichotomique...
2. On utilise les $k-1$ première boules pour faire une recherche dichotomique. Ca isole un intervale de taille $n/2^{k-1}$, qu'on recherche ensuite exhaustivement avec la dernière boule.
3. On lache la première boule aux étages dont le numéro est $i \cdot \sqrt{n}$ pour des valeurs de i croissantes. On isole ainsi un intervale de taille $\sqrt{n}$ contenant la bonne solution, qu'on parcourt exhaustivement avec la deuxième boule.
4. Pour améliorer la technique précédente, il faut déterminer comment le pire cas est atteint. Il est clair que le pire cas est atteint si le seuil est au niveau n. Alors, exactement $2\sqrt{n}$ lancers auront eu lieu. Pour améliorer ça, il faut se dire qu'il n'est pas indispensable de faire au pire $\sqrt{n}$ tests avec chacune des deux boules séparément. Si on fait 1 test avec la première boule, on pourra en faire $\sqrt{2n}-1$ avec la seconde, et c'est bon. Partant de cette observation, l'idée est qu'avec la première boule, on peut faire de plus grand "pas" au début qu'à la fin. En fait, il suffit qu'avec la première boule, le i-ème "pas" soit de longueur $\sqrt{2n}-i$. On est sûrs d'atteindre le sommet de l'immeuble avec un peu moins de $\sqrt{2n}$ lancers, car :

$$\sum_{i=0}^{\sqrt{2n}} \sqrt{2n} - i = n + \sqrt{\frac{n}{2}}$$

   Ensuite, si la première boule se casse lors de son i-ème lancer, alors on a un intervalle de taille $\sqrt{2n}-i$ contenant le seuil, et si on l'explore exhaustivement le nombre total de tests sera $\sqrt{2n}$.

Solution de l'exercice 2

En partant de notre immeuble, on peut lancer la première boule dans l'intervalle $[1;n]$, et on note $f(n,k)$ la solution optimale. Quand on lance la première boule (disons à l'étage i), deux issues sont possibles : ou bien la boule se casse et notre prochain lancer doit être dans l'intervalle $[1;i-1]$, ou bien la boule ne se casse pas et notre prochain lancer doit être dans l'intervalle $[i+1;n]$. Il faut donc, dans le pire des cas, être capable de résoudre les deux sous-problèmes (l'un avec une boule de moins, l'autre avec le même nombre de boules). Clairement :

$$f(n,k) = 1 + \min_{1 \leq i \leq n} \max\Big\{ f(i-1,k-1), f(n-i,k) \Big\}$$

On peut donc très clairement calculer la solution optimale en temps $\mathcal{O}\left(n^2 \cdot k\right)$ et en espace $\mathcal{O}\left(n \cdot k\right)$. Note : il faut bien sûr stocker les valeurs déjà calculées de $f(n,k)$, sinon la complexité sera exponentielle. Il est également possible d'améliorer les complexités temporelles (un des étudiants affirme $\mathcal{O}\left(n \cdot k\right)$), et spatiales ($\mathcal{O}\left(n\right)$ semble facile, mais je ne sais pas si on peut combiner avec l'amélioration précédente).

Solution de l'exercice 3

1. Il suffit de construire le recouvrement petit-à-petit, en ajoutant d'abord l'ensemble S_i qui recouvre le plus de points non couverts. C'est très facile de voir que ce n'est pas optimal (cf. dernière question).
2. Supposons qu'un recouvrement optimal soit formé de l'assemblage de $|I| = k$ ensembles. Notons u_i le nombre de points non-couverts après la i-ème itération de l'algorithme, en fixant $u_0 = n$. Puisqu'on sait que ces u_i points sont totalement couverts par les k ensembles de la solution optimale, il y a forcément au moins un de ces ensembles qui couvre au moins u_i/k points. Le caractère glouton du choix effectué par l'algorithme nous permet alors d'affirmer que : $u_{i+1} \leq u_i - u_i/k$. Il s'ensuit, en déroulant l'expression, que $u_i \leq n(1-1/k)^i$.
   Ensuite, en utilisant l'inégalité de convexité classique $1 - x < e^{-x}$ (valable si x est non-nul, autrement il y a égalité), on trouve : $u_i < ne^{-i/k}$. Lorsque $i = k \ln n$, on trouve $u_t < 1$, et donc il n'y a plus rien à couvrir (l'algorithme a terminé).
3. On pose $B = 0, \ldots, 2^{n+1} - 1$, et on choisit des S_i disjoints tels que $|S_i| = 2^i$. On considère aussi deux ensembles supplémentaires T_0 et T_1 qui contiennent chacun une moitié de tous les S_i. L'algorithme glouton choisi $S_n, \ldots, S_1$, alors que T_0, T_1 est la solution optimale.

Solution de l'exercice 4

1. Nearest-Neighbour : on part d'une ville au hasard, et on va à la ville non-visitée la plus proche, jusqu'à ce qu'on ait fait le tour. On revient alors au point de départ.
2. Repetitive Nearest-Neighbour : on refait la méthode précédente, mais en essayant depuis tous les points de départ possible (et en gardant la meilleure solution bien sûr).
3. Cheapest Link : de façon répétitive, on trace un trait sur la carte entre les deux villes les plus proches, à condition que ce nouveau trait :
   – Ne soit pas le troisième qui touche une ville
   – N'aboutisse pas à la formation d'une boucle dont (au moins) une ville ne ferait pas partie.
   (si la paire de ville considérée ne convient pas, on prend la suivante).
4. Greedy-Insertion : on fabrique une boucle, d'abord réduite à une seule ville, puis on insère dans la boucle la ville qui minimise la taille de la boucle résultante

On peut démontrer que ces techniques offrent une approximation qui est au pire $\mathcal{O}(\log n)$ fois plus grandes que la solution optimale. Il existe par ailleurs une 3/2-approximation qu'on verra plus tard dans l'année, due à Christofides en 1976. Enfin, il existe depuis 2010 (!) un schéma d'approximation polynomial dû à Arora et Mitchell (qui ont gagné le prix Turing pour ça) : pour tout $\epsilon > 0$, il existe un algorithme polynomial en le nombre de ville qui est une $(1+\epsilon)$-approximation.

Solution de l'exercice 5

L'idée est de définir $f(i)$, qui est la meilleure somme partielle obtenue sur un intervalle de la forme $[*; i]$. Il est clair que $\max f(i)$ est la solution du problème. Il suffit de montrer qu'on peut calculer $f(i)$ à partir de $f(i-1)$ et de $A[i]$. Déjà, on observe que $f(i)$ n'est jamais négatif (car on pourrait toujours choisir de prendre la somme vide qui vaut zéro). Ensuite, ou bien $f(i-1) + A[i] \geq 0$, et alors on obtient la meilleure somme partielle terminant sur la i-ème case en étendant la précédente d'une case, ou bien $f(i-1) + A[i] < 0$, et alors $f(i) = 0$. L'algorithme en découle tout seul.

Solution de l'exercice 6

Appelons α et β les deux chaînes de caractères. L'idée est de définir $f(i,j)$, qui est la longueur du plus long *suffixe* commun des *préfixes* de tailles respectives i et j de α et β. Une fois donnée cette information, il est clair que $\max_{i,j} f(i,j)$ est la solution complète. On peut donc déterminer la solution en temps et en espace $\mathcal{O}(|\alpha| \cdot |\beta|)$. Il existe cependant un algorithme plus sophistiqué dont la complexité est $\mathcal{O}(|\alpha| + |\beta|)$.

Solution de l'exercice 7

1. D'abord, la version naïve, en OCaml (avec un paramètre additionnel `shift` qui vaut 0 au début) :

```
1: function MultiSplit(ω, l, shift)
2:     match l with
3:         | [] → []
4:         | i::l' →
5:           let (p,s) = Split(ω,i-shift) in
6:           p :: MultiSplit(s, l', shift+i)
7: end function
```

   Ce sont les appels à Split qui dominent le temps d'exécution. Le premier coûte n, le second $n - a_1$, le troisième $n - a_1 - a_2$, etc. Le coup total est donc :

$$\sum_{i=0}^{n} \left(n - \sum_{j=1}^{i} a_j \right) = n(n+1) - \sum_{i=1}^{n} (n - i + 1) \cdot a_i$$

2. Maintenant, passons à la version améliorée. En fait c'est très similaire aux boules en verre : appelons $f(i,j,k,\ell)$ le coût d'effectuer la découpe de $\omega[i..j]$ par les indices $[a_k; \ldots; a_\ell]$. On a :

$$f(i,j,k,\ell) = (j-i+1) + \min_{s=1..m} \max\Big\{ f(i, a_s, k, s-1), f(a_s, j, s+1, \ell) \Big\}$$

   Et on doit pouvoir conclure en temps $\mathcal{O}(n \cdot m)$.

Solution de l'exercice 8

1. Supposons $\psi \wedge x$ insatisfiable, et prenons une assignation des variables arbitraire (elle ne satisfait pas la formule). Ou bien cette assignation rend x faux, ou bien elle rend x vrai. Dans ce deuxième cas, elle rend ψ faux. Le résultat de la question est la contraposée de cette conclusion.
2. Considérons une assignation $\mathcal{A}$ des variables qui satisfasse ψ. Ou bien $\mathcal{A}$ rend x vrai, ou bien $\mathcal{A}$ rend x faux. De manière équivalente, ou bien $\mathcal{A}$ satisfait $(\psi \wedge x)$ ou bien elle satisfait $(\psi \wedge \overline{x})$. L'algorithme qu'on en déduit est la recherche exhaustive :

```
1: function 3SAT(ψ)
2:     match ψ with
3:     | ∅ → ⊤
4:     | _ → if 3SAT(ψ|x) then ⊤ else 3SAT (ψ|x̄)
5: end function
```

   La complexité dans le pire des cas est donnée par la formule : $T(n) = 2T(n-1) + Poly(n)$, où n désigne le nombre de variables. On aboutit trivialement au résultat annoncé.
3. La formule est obtenue en utilisant les lois de Boole.

```
1: function 3SAT(ψ)
2:     match ψ with
3:     | ∅ → ⊤
4:     | (x ∨ y ∨ z) ∧ ψ' → if 3SAT(ψ|x) then ⊤
5:       else if 3SAT (ψ|x̄y) then ⊤
6:       else 3SAT (ψ|x̄ȳz)
7: end function
```

   Le temps d'exécution obéit à la récurrence : $T(n) = T(n-1) + T(n-2) + T(n-3)$. C'est une récurrence linéaire dont le polynôme caractéristique est $r^3 - r^2 - r - 1$. Sa seule racine réelle est 1.839286755, ce qui donne la complexité du processus.
4. On part d'une assignation des variables dans laquelle il existe un littéral pur non-vrai. On le fait passer à vrai, et la formule reste satisfaite...

```
1: function 3SAT(ψ)
2:     match ψ with
3:     | ∅ → ⊤
4:     | (x ∨ y ∨ z) ∧ (x̄ ∨ u ∨ v) ∧ ψ' → if 3SAT(ψ|xu) then ⊤
5:       else if 3SAT (ψ|xūv) then ⊤
6:       else if 3SAT (ψ|x̄y) then ⊤
7:       else 3SAT (ψ|x̄ȳz)
8: end function
```

   Le temps d'exécution obéit à la récurrence : $T(n) = 2T(n-2) + 2T(n-3)$. C'est une récurrence linéaire dont le polynôme caractéristique est $r^3 - 2r - 2$. Sa seule racine réelle est 1.769292354. Cette technique est due à Monien and Speckenmeyer, et date de 1985 [3].
5. On applique en fait récursivement la technique utilisée sur le littéral x dans la solution précédente à tous les littéraux substitués (cette amélioration est due aux mêmes auteurs). On suppose qu'on donne toujours à la fonction 3SAT une formule sans littéraux purs.

```
1: function FixLiteral(ψ, X)
2:     match ψ with
3:     | ∅ → ⊤
4:     | _ → if ψ|X contient un littéral pur then 3SAT(KillPure(ψ|X))
5:       else let (u ∨ v) ∧ ψ' = ψ|X in
6:         if FixLiteral (ψ|Xu) then ⊤ else FixLiteral (ψ|Xūv)
7: end function
8: function 3SAT(ψ)
9:     match ψ with
10:    | ∅ → ⊤
11:    | (x ∨ u ∨ v) ∧ ψ' → if FixLiteral(ψ, x) then ⊤
12:      else if FixLiteral (ψ, x̄y) then ⊤
13:      else FixLiteral (ψ, x̄ȳz)
14: end function
```

   Notons $T(n)$ le temps d'exécution de 3SAT et $T'(n)$ celui de FixLitteral. On a clairement :

$$T'(n) = \max\Big(T(n-1), T'(n-1) + T'(n-2)\Big)$$
$$T(n) = T'(n-1) + T'(n-2) + T'(n-3)$$

D'où on déduit en substituant la deuxième équation dans la première :

$$T'(n) = T'(n-2) + \max\Big(T'(n-3) + T'(n-4), T'(n-1)\Big)$$

Comme $T'(n)$ est exponentiel en n, on va supposer que $T'(n) = \mathcal{O}(\alpha^n)$. Il résulte de cette hypothèse qu'à l'infini, ou bien $T'(n-3) + T'(n-4)$ est systématiquement plus grand que $T'(n-1)$, ou bien c'est l'inverse.
En fait, $T'(n-3) + T'(n-4) \geq T'(n-1)$ implique que $\alpha \leq 1.325$ (c'est la racine d'une bête équation cubique en α). Cela signifie qu'à l'infini, $T'(n) = T'(n-2) + T'(n-3) + T'(n-4)$. Or résoudre cette récurrence linéaire nous donne $\alpha \geq 1.456$. Contradiction ! Il s'ensuit qu'à l'infini, $T'(n-1)$ est plus grand que $T'(n-3)+T'(n-4)$, et ainsi $T'(n)$ est (asymptotiquement) gouverné par la récurrence $T'(n) = T'(n-1)+T'(n-2)$ dont la solution est $T'(n) = \mathcal{O}(1.6180^n)$. Il s'ensuit qu'à des facteurs polynomiaux près, $T(n) = \mathcal{O}(1.6180^n)$.

Solution de l'exercice 9

Ce qu'il ne faut *pas* faire, c'est, si la salle est libre à partir de l'heure h, accepter la première requête qui se présente dont le début est ultérieur à h, car cela conduit à des solutions sous-optimales. Par exemple, si les requêtes sont $[1;10], [2,4]$ et $[6;8]$, cette méthode conduirait à accepter la première requête et à rejeter les deux autres, tandis qu'en rejetant la première on peut accepter les deux autres.

La bonne solution, c'est de trier les requêtes par date de fin croissante, d'en parcourir la liste dans cet ordre-là, et d'ajouter la requête courante si elle commence après que la salle soit libérée par la requête précédente. Ça se fait donc en temps $\mathcal{O}(n \cdot \log n)$, et on va vérifier que le résultat produit est optimal.

Pour ça, on suppose que n requêtes $[s_1, t_1[, \ldots, [s_n, t_n[$ ont été soumises, et on les suppose triées par heures de fin croissantes. La procédure définie ci-dessus correspond en fait à la définition :

$$\begin{aligned}
\textsc{Greedy}(\emptyset) &= \emptyset \\
\textsc{Greedy}(X) &= \Big(\textbf{let } k = \min X \textbf{ in } \{k\} \cup \textsc{Greedy}(\{i \in X \mid s_i \geq f_k\})\Big)
\end{aligned}$$

Démontrons que Greedy fournit une solution de taille maximale par récurrence sur la taille de son argument. S'il est vide, où s'il est de taille 1, il n'y a rien à faire. Sinon, supposons que $|X| > 2$, et montrons que le choix glouton d'inclure la requête avec la plus petite heure de fin dans la solution est correct. Pour cela on considère une solution optimale du problème, c'est-à-dire un sous-ensemble A de $\{1, \ldots, n\}$ désignant les requêtes acceptées. Appelons également k le plus petit élément de X.

Si $k \notin A$, alors on va fabriquer une autre solution optimale B qui contient k. Considérons la requête de A qui finit le plus tôt, $\ell = \min A$. Par définition, la requête k finit nécessairement *avant* la requête ℓ, donc on peut remplacer ℓ par k dans A sans compromettre ce qui se passe ensuite. Il découle de ceci que $B = A - \{k\} \cup \{1\}$ est une solution non seulement valide, mais aussi optimale puisqu'elle a la même taille que A.

Par hypothèse de récurrence, l'appel récursif à Greedy renvoie une solution optimale (qu'on va appeler S') au problème $X' = \{i \in X \mid s_i \geq f_k\}$, qui est lui-même nécessairement plus petit que X. Il reste à montrer que $S = S' \cup \{k\}$ est bien une solution optimale au problème de départ (X). En fait, si on prend le problème à l'envers, on voit que $B' = B - \{k\}$ est une solution optimale au problème X' : s'il existait une solution de X' strictement plus grande que B' (appelons-la C'), alors $C' \cup \{k\}$ serait une solution valide au problème de départ (X) qui serait strictement plus grande que la solution optimale B (contradiction !). Il s'ensuit que S' et B' ont la même taille, et donc que $S' \cup \{k\}$ a la même taille que B, qui est optimale. L'algorithme retourne donc bel et bien une solution optimale.

Solution de l'exercice 10

1. *(*Next Fit*)* Tout d'abord, si on note $V = \sum a_i$ le volume total de mes affaires, il paraît clair qu'il est impossible d'utiliser *moins* de $\lceil V \rceil$ cartons. Examinons maintenant la solution produite par Next Fit. Plus précisément, examinons deux cartons consécutifs, par exemple C_1 et C_2, ou bien C_3 et C_4, etc. Dans deux cartons consécutifs, le volume des objets présents est strictement supérieur à un (car sinon l'algorithme aurait rangé tous les objets du deuxième carton dans le premier). Ainsi, moins de la moitié de l'espace est gaspillé, et le nombre de cartons utilisé est par conséquent inférieur à $2V$.
   Le cas le pire (ou en tout cas relativement mauvais) est obtenu avec des $4n$ objets de tailles :

$$\left(\frac{1}{2}, \frac{1}{2n}, \frac{1}{2}, \frac{1}{2n}, \frac{1}{2}, \ldots, \frac{1}{2n}, \frac{1}{2}\right).$$

Chaque carton ne contient que deux objets, pour un volume de $\frac{1}{2} + \frac{1}{2n}$. Le nombre ce cartons utilisé est donc $2n$. Par contre, on peut mettre les $2n$ objets de taille $1/2$ dans n cartons, et les $2n$ objets de taille $\frac{1}{2n}$ dans un seul carton...

2. *(*First Fit Decreasing*)* L'idée consiste à partitionner l'ensemble de mes affaires en fonction de leur taille.

$$A = \left\{ \frac{2}{3} < a_i \right\}, \qquad B = \left\{ \frac{1}{2} < a_i \leq \frac{2}{3} \right\},$$
$$D = \left\{ a_i \leq \frac{1}{3} \right\} \qquad C = \left\{ \frac{1}{3} < a_i \leq \frac{1}{2} \right\}$$

On considère ensuite deux cas mutuellement exclusifs :
- Dans la solution DFF, s'il existe au moins un carton ne contenant que des éléments de D (de petits trucs), alors au plus un carton (le dernier) a un taux d'occupation inférieur ou égal à $2/3$. En effet, si les éléments de D du dernier carton n'ont pu être mis dans les cartons précédents, c'est que ceux-ci sont remplis au moins aux $2/3$ (d'où la borne).
- Sinon, s'il n'y a pas de cartons remplis uniquement de bibelots de catégorie D, la solution de DFF est la même que celle de l'instance où on enlèverait tous lesdits bibelots de D (puisque les éléments de D sont rangés après les autres). On va maintenant montrer que si je n'ai pas de bibelots de catégorie D, alors la solution de DFF est optimale. S'il n'y a pas d'objets de catégorie D, alors dans n'importe quelle solution (y compris l'optimale), on observe que :
  - les éléments de A sont tout seuls,
  - il y a au maximum un seul élément de B par carton
  - il y a au plus deux éléments par carton

  La solution optimale est donc obtenue en rangeant d'abord les éléments A dans des cartons à part, puis en mettant les B dans un carton chacun, et enfin en rangeant un C dans chaque carton B, puis en faisant des cartons de 2 C.... Et c'est précisément ce que fait l'algorithme DFF.

Solution de l'exercice 11

Faisons une expérience de pensée. Supposons que les membre portent un dossard indiquant le nom de leur candidat. Supposons qu'une bagarre générale se déclenche, et que les membres du parti des différentes tendances se cognent dessus les-uns les-autres. Supposons aussi l'action symétrique : si un membre x cogne un membre y, alors y cogne aussi sur x. Supposons enfin que tout membre cogné se retrouve à terre, inconscient. Lorsque la situation s'est calmée, il est clair que des supporters d'au plus *une* tendance restent debout, sinon la mêlée reprendrait. Si une tendance avait la majorité absolue, c'est elle qui reste à la fin. Si aucune tendance n'a la majorité absolue, la situation est moins claire, mais en tout cas les délégués restants n'appartiennent pas à la majorité absolue, vu qu'il n'y en a pas. Pour distinguer ces deux situations, le président de séance peut compter le nombre total de dossards de la tendance restante (y compris ceux qui sont à terre), et vérifier s'ils ont la majorité absolue.

Le problème est donc de simuler la bagarre générale. Dans un premier temps, avant de la simuler, nous allons organiser un peu ce pugilat chaotique. Pour cela, le président de séance choisit un premier membre au hasard et le prend avec lui. Il va voir un autre membre au hasard, et les présente. S'ils sont de la même tendance, il embarque les deux membres avec lui, sinon ils s'éliminent mutuellement. Ainsi, le président va promener avec lui un groupe de taille variable de membres qui sont tous d'accord entre eux. Si son groupe se vide, le président le réinitialise avec un nouveau membre choisi au hasard. L'issue fatale est que plus personne n'est debout sauf le groupe autour du président, qui subsiste donc à la fin de la mêlée.

Le président peut simuler pacifiquement cette procédure. D'abord, observons qu'avec $\log_2 n$ bits, il peut stocker la taille de son groupe, et qu'avec $\log_2 k$ bits il peut stocker le candidat de son groupe. Le président met donc tous les membres à la queue-leu-leu devant le confessionnal, puis il exécute la procédure suivante :

```
1: procedure FirstPass
2:     c ← undef
3:     n ← 0
4:     while il reste des membres dans la queue do
5:         faire entrer membre suivant
6:         if n = 0 then
7:             c ← le candidat du membre présent
8:         else
9:             if c = le candidat du membre présent then
10:                n ← n + 1
11:            else
12:                n ← n − 1
13:            end if
14:        end if
15:    end while
16:    return c
17: end procedure
```

Cette procédure révèle le candidat c_0 survivant à la mêlée. Il suffit ensuite de refaire passer tout le monde en leur demandant si leur candidat est bien c_0, et de compter le nombre de réponse (ça tient dans l'espace imparti), pour connaître le résultat final.

Cet algorithme est dû à Boyer et Moore, 1980.

Solution de l'exercice 12

1. Méthode brutale. On énumère tous les sous-ensemble de V (il y en a 2^n), et on teste si chacun d'entre eux est une clique, ce qui prend un temps total $\mathcal{O}\left(n^2 \cdot 2^n\right)$.
2. Il suffit de montrer que l'ensemble d'arêtes est le bon (pas de problème pour les sommets). Si $x \leftrightarrow y$ est une arête du produit, alors ce n'est par définition *pas* une arête de $\overline{G_1} \cup \overline{G_2}$. Trois cas sont possibles :
   – ou bien $x, y \in V_1$, et sous cette condition $(x, y) \notin \overline{G_1} \cup \overline{G_2}$ est équivalent à $(x, y) \in G_2$
   – ou bien $x, y \in V_2$, et sous cette condition $(x, y) \notin \overline{G_1} \cup \overline{G_2}$ est équivalent à $(x, y) \in G_1$
   – ou bien $x \in V_1$ et $y \in V_2$ (on peut le supposer sans perte de généralité puisque le graphe est non-orienté). Sous cette condition, $(x, y) \notin \overline{G_1} \cup \overline{G_2}$ est toujours vrai.
3. a) Par définition, un cographe peut être représenté par un *arbre de dérivation*, dont les feuilles sont étiquetées par des sommet (tous différents), et les noeuds sont soit étiquetés par "union", soit par "complémentaire". La fonction suivante associe le cographe à l'arbre de dérivation (qu'on appelle un "coarbre") :

$$\begin{aligned} \rho(x) &\to (\{x\}, \emptyset) \qquad \text{si } x \text{ est une feuille} \\ \rho(T_1 \cup \cdots \cup T_n) &\to \rho(T_1) \cup \cdots \cup \rho(T_n) \\ \rho\left(\overline{T}\right) &\to \overline{\rho(T)} \end{aligned}$$

   On montre qu'on peut construire l'arbre de dérivation en temps polynomial à partir du graphe. Soit on a affaire à un sommet isolé, et c'est assez facile. Soit on a affaire à un graphe non-connexe (testable en temps polynomial), et on attaque chaque composante connexe séparément tout en mettant un noeud "union" à la racine de l'arbre. Soit le graphe complémentaire est non-connexe, et on se ramène au cas précédent en mettant un noeud "complémentaire" à la racine de l'arbre. Enfin, si le graphe complémentaire est connexe, c'est que le graphe n'est pas un cographe.

   b) On montre qu'on peut se passer des noeuds "complémentaire" à condition de s'autoriser des noeuds "produit". On étend la fonction ρ par :

$$\rho(T_1 \bowtie \cdots \bowtie T_n) \to \rho(T_1) \bowtie \cdots \bowtie \rho(T_n)$$

   On peut alors "réécrire" l'arbre (de manière non-déterministe parce que c'est plus pratique) en appliquant *ad libitum* les quatre règles suivantes :

$$\begin{aligned} \overline{T_1 \cup \cdots \cup T_n} &\to \overline{T_1} \bowtie \cdots \bowtie \overline{T_n} \\ \overline{\overline{T}} &\to T \\ \overline{x} &\to x \qquad \text{si } x \text{ est une feuille} \\ (S_1 \cup \cdots \cup S_n) \cup T_2 \cup \cdots \cup T_n &\to S_1 \cup \cdots \cup S_n \cup T_2 \cup \cdots \cup T_n \end{aligned}$$

On se convainc d'abord que l'application de ces règles termine. En effet, elle font soit diminuer strictement le nombre de noeuds qui sont du même type que leur parent, soit elles poussent les négation d'un étage vers le bas (ce qui ne peut pas se faire infiniment souvent, même si l'argument est informel), soit elle font diminuer le nombre total de négations. Il est facile de vérifier que l'application de chacune des règles séparément ne modifie pas la valeur de ρ sur l'arbre. Enfin, une fois qu'on a fini de les appliquer, il n'y a plus de noeud "complémentaire" dans l'arbre.

c) Une fois qu'on a un coarbre "union/produit", il n'est pas dur de déterminer la taille de la plus grosse clique dans le cographe correspondant :

$$\begin{aligned}
\textsc{MaxClique}(x) &= 1 \qquad \text{si } x \text{ est une feuille} \\
\textsc{MaxClique}(T_1 \cup \cdots \cup T_n) &= \max\Big\{\textsc{MaxClique}(T_1), \ldots, \textsc{MaxClique}(T_n)\Big\} \\
\textsc{MaxClique}(T_1 \bowtie \cdots \bowtie T_n) &= \sum_{i=1}^{n} \textsc{MaxClique}(T_i)
\end{aligned}$$

Le seul point non-trivial est de vérifier la troisième égalité. En fait, comme l'opération "produit" est trivialement associative, il suffit de démontrer que

$$\textsc{MaxClique}\left(\overline{\overline{T_1} \cup \overline{T_2}}\right) = \textsc{MaxClique}(T_1) + \textsc{MaxClique}(T_2)$$

pour conclure par récurrence. Pour cela, on va construire explicitement une clique de la bonne taille sur le produit. Notons $G_i = (V_i, E_i) = \rho(T_i)$, et $C_i \subset V_i$ une clique de taille maximale dans G_i. D'après le résultat qu'on a montré sur le produit, chaque sommet de C_1 est relié à tous les sommets de G_2 (donc en particulier à tous ceux de C_2), et vice-versa. Ceci prouve que $C_1 \cup C_2$ est une clique dans le produit. Prouvons maintenant qu'elle a la taille optimale, et pour cela prenons une clique de taille maximale dans le produit. On voit que $C'_i = C \cap G_i$ sont deux cliques "induites" sur les sous-graphes séparés, et ces dernières sont forcément plus petites que les cliques maximales sur les deux sous-graphes. On peut conclure à partir de là.

Solution de l'exercice 13

Plusieurs preuves sont possible. La plus "simple" consiste à observer que chaque fois qu'Hercules coupe une tête, cela fait décroître l'ordre *multi-ensemble emboité* sur l'arbre (cette relation d'ordre a été introduite en 1979 par Dershowitz et Manna [1]). Ensuite, comme cet ordre est bien fondé, il ne peut pas y avoir de chaînes infinies décroissantes, et la terminaison est établie. L'idée d'utiliser des relations bien fondée pour prouver la terminaison a été suggérée par Floyd en 1967 [2].

Mais on va utiliser une approche plus directe, qui fonctionne en deux temps. On va d'abord démontrer qu'il est possible, indépendemment de la valeur de i, de transformer n'importe quel sous-arbre de hauteur 2 en un sous-arbre de hauteur 1, sans modifier le reste de l'arbre. Ensuite, on utilisera cette propriété pour conclure par induction.

- Considérons un sous-arbre de hauteur 2. Il y a une "racine", des "noeuds internes" de hauteur 1, des feuilles de hauteur 2, et éventuellement des feuilles de hauteur 1. Lorsqu'on coupe une feuille de hauteur 2, le noeud interne qui est son père est cloné i fois. On s'intéresse au nombre maximal de feuilles qui descendent du même noeud interne, $n_{\max}$. Il peut y avoir plusieurs noeuds internes (disons k) ayant précisément ce nombre de feuilles. Cependant, couper une de ces feuilles fait baisser de un le nombre de noeuds interne ayant $n_{\max}$ feuilles, en échange de la création de i noeuds internes ayant $(n_{\max} - 1)$ feuilles. Cela prouve qu'en k coupes, on peut faire baisser de 1 le nombre maximal de feuilles descendant du même noeud interne. Clairement, la taille de l'arbre résultant de ces k coupes, même si elle est plus grande, est toujours finie (on a rajouté $i + (i+1) + \cdots + (i+k)$ noeuds internes). Cela prouve qu'en un nombre fini d'étapes, on peut se ramener à la situation où tous les noeuds internes (qui seront certes en très grand nombre) n'ont plus de feuilles. On a donc fait baisser de 1 la hauteur de notre sous-arbre
- Etant donné cette procédure qui transforme un sous-arbre de hauteur deux en un sous-arbre de hauteur un, on va maintenant conclure. En fait, si l'arbre de départ possède m noeuds internes, moins de m application de la procédure sont nécessaire. En effet, si on repère toutes les racines de sous-arbre de hauteur deux (il y en a nécessairement moins de m), et qu'on applique la procédure à chacune d'entre elles, on fait baisser la hauteur totale de l'arbre de un. Ceci permet de conclure qu'on peut détruire l'arbre en un temps fini.

Références

[1] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22 :465–476, August 1979.

[2] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32) :1, 1967.

[3] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2n steps. *Discrete Appl. Math.*, 10 :287–295, March 1985.